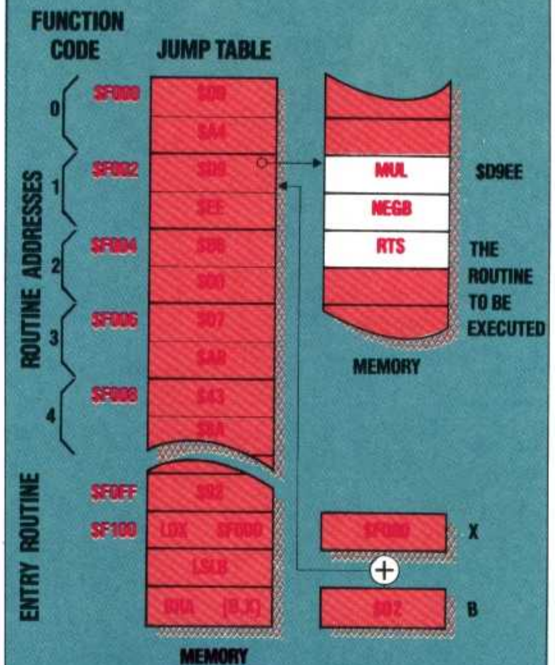
character passed in A to the screen at the current cursor position, unless that character is a carriage return (ASCII 13), in which case it will clear the rest of the line and position the cursor at the beginning of the next line. The cursor is represented by the underline character ('_') in this example.

```
SPACE   EQU 32          ASCII code for space
CR      EQU 13          ASCII code for carriage return
HOME    EQU $E000       Start of screen memory
LENGTH  EQU 1024        Size of screen memory (16 lines × 64
                        characters = 1024)
CURSOR  EQU $E400       $E400 and $E401 together point to
                        the current address of the cursor in
                        screen memory area
        ORG $1000
CURCHR  FCB 95          Underline character (ASCII 95)
***Subroutine to clear screen***
        LDA #SPACE      Space character in A
        LDX #HOME       Point cursor to beginning of screen
        STX CURSOR      Store current cursor position at
                        CURSOR (i.e. $E400, $E401)
        LDB #LENGTH     Size of screen in B
LOOP1   STA [CURSOR]    Store a space in current cursor
                        position
        INC CURSOR      Increment cursor position
        DECB            Decrement amount of screen
                        memory remaining between cursor
                        position and end of screen memory
        BGT LOOP1       Next space, until no more screen
                        memory remains
        STX CURSOR      Cursor back to home position
        LDA CURSOR      ASCII code of cursor character in A
        STA [CURSOR]    Store cursor character in current
                        cursor position
        RTS
***Subroutine to display character in A, if displayable***
        CMPA SPACE      Space is the first printable character
                        in ASCII
        BLT NOTP        If accumulator contains ASCII value
                        less than 32, this is non-printable, so
                        GOTO NOTP
        STA [CURSOR]    Store in current cursor position
        INC CURSOR      Increment cursor position
CHKEOS  LDX #HOME       Check for end of screen
        LEAY #LENGTH,X  End of screen in Y
        CMPY CURSOR     If cursor position exceeds end of
                        screen then . . .
        BGT FINISH      we have reached the end of the
                        screen, so GOTO FINISH
***Subroutine to scroll screen***
SCROLL  LEAY 64,X       Y is one line length from X (end of
                        screen memory)
        LDB #LENGTH     Calculate amount to scroll
        SUBB #64        Subtract 64 from length
LOOP2   LDA ,Y+         Move characters back one line (notice
                        auto-increment — see page 618)
        STA ,X+
        DECB
        BGT LOOP2       Loop until scrolling complete
        LDD CURSOR      Cursor to start of last line
        SUBD #64
        STD CURSOR
        BRA FINISH
***Subroutine to check for carriage return***
NOTP    CMPA #CR        Is this non-printable character a
                        carriage return?
        BNE FINISH      Ignore if not
        LDD CURSOR      You can work out how this gives the
        ANDB #%11100000 start of the next line (notice binary
                        AND-mask)
        ADDD #64
        STD CURSOR
        BRA CHKEOS      Check if end of screen
FINISH  LDA CURSOR      Cursor character in A
        STA [CURSOR]    Store in current cursor position
        RTS
```

The Jump Table



The jump table in this example is a list of 128 two-byte address pointers located between $F000 and $F0FF. Each of these pointers contains the start address of a routine somewhere in memory. To execute any of these routines we need only load the B accumulator with a function code ($01, for instance) which identifies the desired routine (located at $D9EE in this example) and then JSR to the so-called 'entry routine', start address $F0FF here. We assume that these routines are in ROM (because they are part of some ROM-based software such as the operating system) so we will be able to look up the function code and the entry routine start address in the programmer's manual.

The entry routine multiplies the function code by two, and uses it as an offset to the table start address to find the desired routine's address pointer: the pointer to routine $01 is located at $F002, for example (=$F000+2×$01), routine $02's pointer is at $F004 (=$F000+2×$02), and so on. The pointer is then used by the entry routine in an indirect branch instruction to pass control to the actual routine at $D9EE. Notice that the entry routine branches to (rather than calls) the execution routine, so that when RTS is encountered, control will pass back to the point in the program from which the entry routine was first called.

The advantage of a jump table (especially when used with an entry routine) is that it allows programmers to redesign and relocate the routines that it addresses, while still permitting programs written before such revisions to run on the new system: the function codes and the address of the entry routine are kept constant throughout the life of the system, but the contents of the jump table address pointers (and even the location of the jump table itself) may change at will.